Attorney Docket No. 06502.0269

UNITED STATES PATENT APPLICATION

OF

PETER C. JONES,

ANN M. WOLLRATH

AND

ROBERT W. SCHEIFLER

FOR

METHOD AND SYSTEM FOR DYNAMIC PROXY CLASSES

LAW OFFICES
FINNEGAN, HENDERSON, FARABOW, GARRETT, & DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

BACKGROUND

Field of the Invention

This invention generally relates to a data processing system and, more particularly, to a method and system in an object-oriented data processing system for generating proxy classes at runtime that implement a list of interfaces specified at runtime.

Related Art

Many computing systems use object-oriented programming to better accommodate the increasing complexity and cost of large computer programs. Object-orient programming languages have grown to be widely used due to their programming power and efficiency. In an object-oriented computing system, source code written in an object-oriented programming language, such as the Java™ programming language, typically contains a number of "classes," such as Java classes. A class is a template for creating objects that are "instances" of the class and are created from the template. A class contains both data members that store data and function members (or "methods") that act upon the data. Consequently, the class of an object defines its data and behavior, and the methods (function members) are typically used to both get and set the values for the data members in the class.

Additionally, object-oriented programming languages such as the Java programming language utilize "interfaces," which are simply lists of methods. An interface is similar to a class but typically only has declarations of its methods and typically does not provide implementation of those methods. An interface specifies methods supported by classes that implement the interface and declares what those methods should do. Classes, instances, interfaces and other

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

aspects of object-oriented programming languages are discussed in detail in "The Java Programming Language", 2nd Ed., Ken Arnold, James Gosling, Addison-Wesley, 1998, which is incorporated herein by reference. For further description of the Java language, refer to "The Java Language Specification," James Gosling, Bill Joy, Guy Steele, Addison-Wesley, 1996, which is also incorporated herein by reference.

When the source code for classes, interfaces and other aspects of an object-oriented programming language are created, they are usually written by a user and then compiled by a compiler on a computer so that the computer may run the source code. The source code is written and then compiled at a stage referred to as "compile-time." After the source code has been built and compiled, a computer system may then execute the source code at "runtime."

Since a class typically has to be built offline and then compiled, the interfaces implemented by that class must typically be known while the class is being built before runtime. Conventionally, to implement an interface by a class, the methods of the interface are determined during source code development before compile-time. For example, section 4.5 of Arnold and Gosling's "The Java Programming Language" describes an approach called "forwarding" to make a class use a second class for the implementation of one of its interfaces. This approach is simply to define every single method of the interface in the first class to delegate to the corresponding method in the second class. However, section 4.7 characterizes this approach as "tedious to implement and error-prone," which is further demonstrated by the fact that compile-time tools have been developed to automate the process of generating these forwarding methods.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

Sometimes the interfaces that a user would like to implement may not be known or may be changing during the runtime of the program. As a result, it would be desirable to have a method and system that avoids the need to generate before runtime a class that implements a list of interfaces and avoids the need to write and compile code for each method of an interface being implemented.

Conventionally, to implement a single functionality on multiple interfaces of varying types, code needs to be created for each interface to carry out the functionality. It would also be desirable to not require the writing of specialized code for each one of multiple types of interfaces to facilitate a single functionality.

SUMMARY

Methods and systems consistent with the present invention provide a proxy class that is dynamically generated at runtime and that implements a list of interfaces specified at runtime. Method invocations on an instance of the proxy class are encoded and dispatched uniformly to another object that handles the method invocation. In accordance with the present invention, the proxy class generated at runtime does not require the implemented interfaces and their methods to be known before runtime, *i.e.*, during development or at compile-time. Since no pre-generation of the class before runtime is required, the interfaces desired to be used may change during runtime before the creation of the proxy class. Additionally, the proxy class may provide uniform implementation of multiple types of interfaces without requiring specialized code for each one.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

In accordance with methods and systems consistent with the present invention, a client may invoke a method of an interface implemented by the proxy class to an instance of the proxy class, and the proxy class instance encodes and dispatches the method to an invocation handler object associated with the proxy class instance that processes the method invocation and returns the result to the client via the proxy class instance.

In accordance with methods and systems consistent with the present invention, a method in a data-processing system generates at runtime a proxy class that implements one or more interfaces specified at runtime having methods, and creates an instance of the proxy class. Further, the proxy class instance receives a request to process a method of an interface, and dispatches the request to an invocation handler object that processes the request for the method.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 depicts a data processing system suitable for use with methods and systems consistent with the present invention;

Figure 2 depicts a proxy class and proxy class instance that implement the interfaces in accordance with methods and systems consistent with the present invention;

Figure 3 depicts a flowchart illustrating steps used in a method for generating a proxy class, a proxy class instance, and an invocation handler at runtime consistent with the present invention;

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

Figure 4 depicts a flowchart illustrating steps used in a method for creating and returning a proxy class given a list of interfaces consistent with the present invention; and

Figure 5 depicts a flowchart illustrating the steps used in a method for making a method invocation on an instance of a proxy class generated at runtime in accordance with methods and systems consistent with the present invention.

DETAILED DESCRIPTION

In an object-oriented data processing system, methods and systems consistent with the present invention provide a proxy class, dynamically generated at runtime, that implements a list of interfaces specified at runtime. A method invocation through an interface on an instance of the class is encoded and dispatched uniformly regardless of the type of interface to an invocation handler object that performs the invocation of the requested method. Thus, in this manner, multiple interfaces may share the same code, which reduces wasteful duplication.

The dynamic generation of the proxy class at runtime and the specification at runtime of the list of interfaces and their methods supported by the proxy class allow the interfaces to be chosen after compile-time, but before the generation of the proxy class. Since the proxy class does not need to be created before compile time, the list of interfaces implemented by the proxy class does not need to be known at the time the source code is written, *i.e.*, during development. Thus, a client that uses the proxy class to invoke methods on one or more interfaces is independent of the interfaces, so if different interfaces are selected, the client does not need to rebuild the proxy class. Moreover, as new interfaces are become available, the client may utilize them without modification

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

The proxy class provides uniform access to the methods of the multiple types of interfaces implemented by the proxy class without regard to the type. If a client wishes to make a method invocation for a method of an interface implemented by the proxy class, the client may send the method invocation to an instance of the proxy class. The proxy class instance automatically encodes and dispatches a method invocation of a method on an interface implemented by the proxy class instance to an invocation handler object that automatically handles the request and returns the result. The invocation handler returns the result to the proxy class instance and then back to the client that made the original method invocation request.

The invocation handler is an object created by the client that is associated with the proxy class instance and processes the request received from the proxy class instance. The invocation handler is created to be application-specific and can be formed in a wide variety of ways depending on the application desired by the client.

For an example use of a dynamic proxy class, consider the following. A client may want to register to receive notification of events from a number of event generators, such as a windows manager and a device driver. However, each event generator typically may use a different interface and the programmer may not know at compile time what the interfaces may contain or even which interfaces the client may need at runtime. The dynamic proxy class in accordance with methods and systems consistent with the present invention solves this problem.

In using the dynamic proxy class, the client, at runtime, specifies the interfaces in which it is interested. That is, the client may receive an indication through the user interface or some other runtime mechanism which causes it to want to utilize a number of interfaces. So, the client invokes, for example, a Java Class Library (described below) call using a name or other identifier

LAW OFFICES
FINNEGAN, HENDERSON, FARABOW, GARRETT, & DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

to obtain a class object representative of each desired interface. Then, the client utilizes another call to create a dynamic proxy class, which supports the interfaces. Lastly, the client instantiates the proxy class by providing it an invocation handler. Once the proxy class is instantiated, the instance of the proxy class implements the interfaces and can be registered, for example, with each of the event generators. Since the proxy object supports each of the event generator interfaces, each event generator recognizes the proxy object and can invoke the methods on the interface for that event generator. Furthermore, for every method invoked, the same invocation handler is invoked, thus needless duplication of code is prevented. This invocation handler performs common functionality for each of the methods. For instance, for each method invoked, representing the occurrence of an event, the invocation handler may log the event into a file. In this manner, the client need not know of the interfaces before runtime and can have common functionality used for a number of interfaces.

Although the example is described as using the proxy class by an entity other than the creator, one skilled in the art will appreciate that it may be beneficial for a client to both create and use the proxy class.

As another example, dynamic proxy classes may be applied to remote method invocations ("RMI") that allow objects executing on one computer to invoke method of an object on another computer. If a proxy class is to be used for the purpose of RMI on a remote server, the interfaces implemented by the dynamic proxy class might be a list of the methods available on the remote server, and the invocation handler might perform the function of forwarding the call to the location of a remote object. In this manner, the dynamic proxy class can be created using interface specifications received from a remote server at runtime, and the interface

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

specifications need not be known beforehand at compile-time. In conventional systems, the proxy class is typically built offline, compiled and then downloaded before runtime, thereby reducing runtime flexibility.

Figure 1 depicts a data processing system 100 suitable for use with methods and systems consistent with the present invention. Data processing system 100 includes computer 101 interconnected to a network 114 such as the Internet, a wide area network ("WAN") or a local area network ("LAN"). Computer 101 includes a central processing unit (CPU) 102, a main memory 104, and a secondary storage device 106 interconnected via bus 108. Additionally, client computer 101 includes a display 210, and an input device 212. The computer 101 also includes a client 116, which may be any object, computational entity or program. The client 116 need not necessarily reside in memory 104 of computer 101 (*e.g.*, it may be located elsewhere, possibly on another computer). Although only one computer 101 is shown, there may be many additional computers connected to network 114.

Main memory 104 also contains a virtual machine 118 ("VM"), such as a Java VM, which acts as an abstract computing machine, receiving instructions from programs in the form of byte codes and interpreting these byte codes by dynamically converting them into a form for execution, such as object code, and executing them. Java virtual machines are described in detail in "The Java Virtual Machine Specification," Lindholm and Yellin, Addison-Wesley, 1997, which is incorporated herein by reference. The memory also contains a runtime system, such as a Java Runtime Environment ("JRE") 120, which further contains class file libraries 124 which define classes used by programs running on the JRE. The class file libraries 124 contain definitions for "Class.forName" and "getProxyClass" methods which will be discussed below.

LAW OFFICES
FINNEGAN, HENDERSON, FARABOW, GARRETT, & DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

The Class.forName function and other aspects of Java class libraries are described in detail in " The Java Class Libraries: An Annotated Reference," Chan and Lee, Addison-Wesley, 1997, which is incorporated herein by reference. Although Java components are discussed, methods and systems consistent with the present invention may utilize other types of components.

Figure 2 depicts a proxy class 202 in a VM 118 which is generated at runtime and implements interfaces specified at runtime, such as interfaces 206 and 208, which include methods not shown on this diagram. In one implementation consistent with the present invention, another class (not shown) is used by the client 116 to generate the proxy class 202, and the client creates the proxy class by calling the getProxyClass method of the other class and passing representations of the desired interfaces as arguments to that method. Further, the client 116 creates an instance 204 of the proxy class 202 by giving the invocation handler 122 (described below) as an argument to the proxy class 202. The proxy class instance 204 can then be used by the client 116 or other entity to invoke a method of one of the interfaces implemented by the proxy class instance 204, such as interface 206 or 208.

The main memory 104 also contains an invocation handler 122 that is an object created by the client 116. The invocation handler 122 is created to be application-specific and may vary depending on the purpose of the proxy class 202. For example, in the event listener scenario, the invocation handler 122 performs the function of logging events to a file. As stated, the client 116 creates the proxy class instance 204 by passing the invocation handler 122 as an argument and thus is associated with the invocation handler 122 upon creation. When the proxy class instance 204 receives a method invocation request from a client 116, the proxy class instance 204 encodes and dispatches the method request to the invocation handler 122. The invocation handler 122

LAW OFFICES
FINNEGAN, HENDERSON, FARABOW, GARRETT, & DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

processes method invocations made to the proxy class instance 204 and returns a result to the proxy class instance to be returned to the client 116.

Although aspects of the present invention are described as being stored in memory or a VM, one skilled in the art will appreciate that these aspects can also be stored on or read from other types of computer-readable media, such as secondary storage devices, like hard disks, floppy disks, or CD-ROM; a carrier wave from a network, such as the Internet; or other forms of RAM or ROM either currently known or later developed. Sun, Sun Microsystems, the Sun logo, Java™, and Java™-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Figure 3 is a flowchart illustrating steps used in a method for generating a proxy class 202 and a proxy class instance 204. It should be appreciated that, in one implementation consistent with the present invention, the steps occur at runtime, *i.e.*, after development of the source code and compile-time. A client 116 generating a proxy class 202 at runtime determines the interfaces that will be supported by the proxy class 202 (step 302), and this determination may be made at runtime by the client 116 and will depend on the purposes for which the proxy class will be used, such as for event listener applications as mentioned above. In the example shown in Figure 2, the proxy class 202 supports the interfaces 206 and 208.

In one implementation consistent with the present invention, once the interfaces 206 and 208 to be implemented are determined, using the names of the interfaces, a client 116 may call a function such as the Class.forName function to obtain the "Class Objects" of the desired interfaces (step 304). The Class Objects are representations of the interfaces 206 and 208 and are used as the arguments for the getProxyClass method. Given the names of the desired interfaces

LAW OFFICES
FINNEGAN, HENDERSON, FARABOW, GARRETT, & DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

206 and 208, the Class.forName method returns the Class Objects of the requested interfaces (step 306).

Once the client 116 has the Class Objects of the desired interfaces 206 and 208, it may generate a proxy class 202 for the interfaces by calling the getProxyClass method using the interface's Class Objects as arguments (step 308). The getProxyClass method returns a proxy class 202 that implements the interfaces 206 and 208 (step 310). To this end, the getProxyClass method generates custom code for each method of the interfaces that will encode and dispatch a method invocation to the invocation handler 122 including the arguments. This process is described below in further detail in Figure 4. The client also creates the invocation handler 122 that will carry out the desired functionality of a call to an instance 204 of the proxy class 202 (step 312). The client then creates a proxy class instance 204 by calling the constructor of the proxy class 202 with the invocation handler 122 as an argument (step 314).

Figure 4 depicts a flowchart illustrating steps used in a method for returning a proxy class given a list of interfaces. Consistent with the present invention, the steps illustrated in Figure 4 are performed by the getProxyClass method invoked by the client 116. When the client 116 invokes this method with the Class Objects for the interfaces 206 and 208 to be implemented, the method verifies the arguments to make sure that they are valid (step 402). For each method of the interfaces 206 and 208 received as arguments, it generates byte codes that will package parameters and pass them to the invocation handler 122 (step 404). Then, the getProxyClass method creates the code for the constructor that will create an instance 204 of the proxy class 202

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

(step 406). Next, it stores the constructor and the byte codes in a class file format (step 408). Using the class file, the getProxyClass method then requests the VM 118 to define the class so that it is a valid class that is recognizable by the VM so that the VM will allow clients to instantiate it (step 410). Finally, the getProxyClass method returns the proxy class 202 (step 412).

Figure 5 is a flowchart illustrating steps used in a method for making a method invocation on an instance 204 of a proxy class 202 generated at runtime in accordance with methods and systems consistent with the present invention. Once a caller has access to the proxy class instance 204, the caller may wish to invoke a method (step 502). For example, in the event listener example, an event generator may wish to log a window manager event or other type of event so it invokes a method on one of the interfaces 206 or 208 supported by the proxy class instance 204. Upon invocation, the proxy class instance 204 encodes and dispatches the method invocation to the invocation handler 122 (step 504). The invocation handler 122 then processes the method invocation using its predetermined functionality and obtains the result (step 506). For example, in this step, the invocation handler 122 may log an event reflected by the parameters. The invocation handler 122 then returns the result to the proxy class instance 204 (step 508) which then returns the result back to the caller (510).

As a result, methods and systems in accordance with the present invention allow a proxy class to be generated at runtime that can implement interfaces specified at runtime thereby avoiding a need for pre-generation of the proxy class and predetermination of the interfaces before runtime. They also facilitate the processing of method invocations to a single object that uniformly handles method invocations of varying types of multiple interfaces.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

The following code contains portions of commented code that outlines a class that may be used for the creation of a proxy class written in the Java programming language. Proxy classes, as well as instances of them, may be created using the static methods of the this class (note that the javadoc in the code presented here is not specification-complete):

```
package java.lang.reflect;

/**
* The Proxy class provides static methods for dynamically creating
* a class that implements a specified list of interfaces. Method
* invocations through those interfaces on instances of the dynamically
* created proxy class are encoded and dispatched to a special invocation
* handler object supplied during creation of the proxy instance. The
* invocation handler is an instance of the InvocationHandler interface.
*
* To create a proxy for some interface Foo:
*
*       InvocationHandler handler = new MyInvocationHandler(...);
*       Class proxyClass = Proxy.getProxyClass(
*               Foo.class.getClassLoader( ), new Class[ ] { Foo.class });
*        Foo f= (Foo) proxyClass.
*               getConstructor(new Class[ ] { InvocationHandler.class}).
*               newInstance(new Object[ ] { handler });
*
* or more simply:
*
*       Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader( ),
*                                            new Class[ ]  {Foo.class}, handler);
* A proxy instance created via reflection on the proxy class or via the
* new Proxy Instance static method can be cast to any of the interfaces
* specified when the proxy class was created.

* When an interface method is invoked on the proxy, the method call is
* dispatched t the invocation handler's invoke method, passing the
* proxy instance, the java.lang.reflect.Method object for the interface
* method, and an array containing the arguments passed:
*
*       handler.invoke (proxy, method, args);
*
```

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

```
* The handler processes the method invocation as appropriate and the
* result it returns will then be returned to the caller of the proxy.
*/
public class Proxy implements java.io.Serializable {
    protected InvocationHandler h;

    protected Proxy (InvocationHandler h) {
      this .h = h;
    }
    /**
    * Returns a class that implements the specified interfaces.
    * A class is dynamically created if one does not already exist.
    *
    * @param     loader the class loader to define the proxy class in
    * @param     interfaces the list of interfaces for the proxy class to implement
    * @return    the specified dynamic proxy class
    * @throws    IllegalArgumentException if the arguments do not identify
                 a valid proxy class
    */
    * public static Class getProxyClass (ClassLoader loader, Class [ ] interfaces)
    throws IllegalArgumentException
    {
        // returns proxy class that implements interfaces
    }
    /**
    * Returns an instance of a proxy class that implements the
    * specified interfaces and dispatches method invocations to
    * the specified handler.  This method is equivalent to:
    *
    *       Proxy.getProxyClass (loader, interfaces).
    *             getConstructor (new Class[ ] {InvocationHandler.class }).
    *             newInstance (new Object { ] {  handler });
    * @param     loader the class loader to define the proxy class in
    * @param     interfaces the list of interfaces for the proxy class
                 to implement
    * @param     handler the handler to dispatch proxy invocations to
    * @return    the specified dynamic proxy class
    * @throws    IllegalArgumentException if the arguments do not
    *            identify a valid proxy class
```

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

```
    */
    public static Object newProxyInstance (ClassLoader loader,
                                           Class [ ] interfaces,
                                           InvocationHandler handler)

        throws IllegalArgumentException
    {
        // returns instance of specified proxy class
    }

    /**
    * Returns true if and only if the specified class was dynamically
    * generated to be a proxy class using the getProxyClass
    * method or the newProxyInstance method.
    *
    * @param     cl class to test
    * @return    true if the class is a proxy class and false otherwise
    */
    public  static boolean isProxyClass (Class cl)  {
        // returns true if class is a dynamic proxy class
    }

    /** Returns the invocation handler of the specified proxy instance.
    *
    * @param     proxy the proxy instance to return invocation handler of
    * @return    the invocation handler of the proxy instance
    * @throws    IllegalArgumentException if the argument is not a
    *            proxy instance
    */
    public static InvocationHandler getInvocationHandler (Object proxy)
        throws IllegalArgumentException
    {
        // returns invocation handler for proxy instance
    }
}

/*  InvocationHandler is used in conjunction with a proxy instance.  When a
* method is invoked on a proxy instance, the method invocation
* is encoded and dispatched to the invoke method of its associated
* InvocationHandler instance.
```

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

```
*/
public interface InvocationHandler {
        /**
        * Processes a proxied method invocation and returns a result.
        */
        Object invoke(Object proxy, Method method, Object[ ] args)
                throws Throwable;
}
```

The foregoing description of an implementation of the invention has been presented for purposes of illustration and description. It is not exhaustive and does not limit the invention to the precise form disclosed. Modifications and variations are possible in light of the above teaching or may be acquired from practicing of the invention. The scope of the invention is defined by the claims and their equivalents.